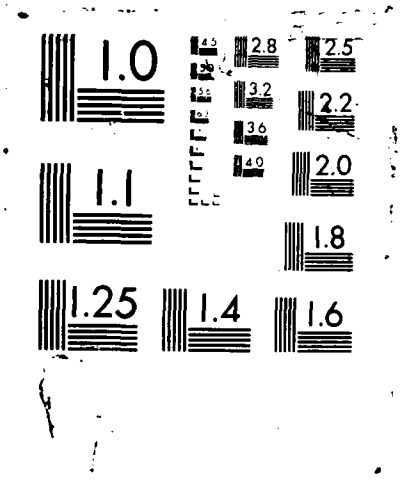
AD-A193 471 A CLASSIFICATION OF AUTOMATIC PROGRAM SYNTHESIS SYSTEMS 1/1
(U) NEVADA UNIV LAS VEGAS DEPT OF COMPUTER SCIENCE AND
ELECTRICAL.. T A GROSS ET AL. 19 JAN 88 CSR-88-03
UNCLASSIFIED ARO-24960.8-MA DAAL03-87-G-0004 F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

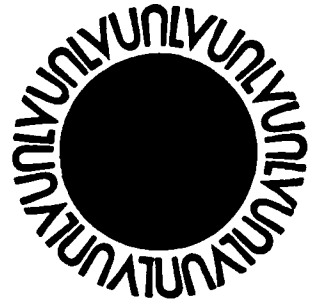
ARO 24960.8-M

A Classification of Automatic Program Synthesis Systems

Todd A Gross
Thomas A Nartker*
Department of Computer Science and Electrical Engineering
University of Nevada, Las Vegas

January 19, 1988

Department of Computer Science and Electrical Engineering



University of Nevada, Las Vegas
Las Vegas, Nevada 89154

AD-A193 471

REPORT DOCUMENTATION PAGE

ORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
:URITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
CLASSIFICATION/DOWNGRADING SCHEDULE	
FORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): ARO 24960.8-MA
ME OF PERFORMING ORGANIZATION: Univ. of Nevada	6b. OFFICE SYMBOL (If applicable)
	7a. NAME OF MONITORING ORGANIZATION: U. S. Army Research Office
DDRESS (City, State, and ZIP Code): Las Vegas, Nevada 89154	7b. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: U. S. Army Research Office	8b. OFFICE SYMBOL (If applicable)
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DAAL03-87-G-0004
8c. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): A Classification of Automatic Program Synthesis Systems

12. PERSONAL AUTHOR(S): Todd A. Gross and Thomas A. Nartker

13a. TYPE OF REPORT	13b. TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical		January 19, 1988	14

16. SUPPLEMENTARY NOTATION: The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Computers, Automatic Program Synthesis Systems

19. ABSTRACT

A great deal of attention has been paid lately to improving the process of developing software. Most of this attention has been directed to the development of *tools* to perform tasks for the user (e.g. editors, debuggers) and *environments* to integrate these tasks. This paper, however, is about another method of streamlining software development: creating programs to generate the software for us. That is, we tell the program what type of software we want to generate and it generates the software for us. This process has been given many names.

In this paper, we shall refer to this process as *automatic program synthesis*, or APS.

In Section 2 we define the term automatic program synthesis. Section 3 gives a set of classifications of various APS systems. Section 4 gives the authors' conclusions about existing APS systems and about the field of automatic program synthesis in general. Section 5 comments on the related yet distinct area of application generation.

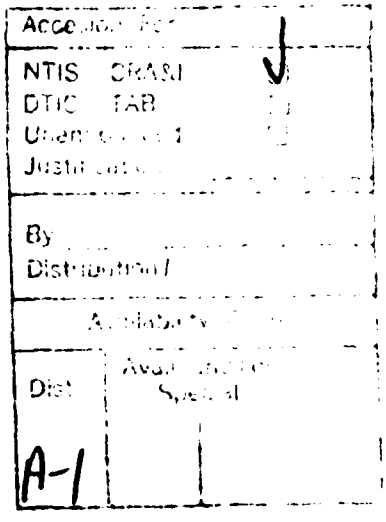
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

A Classification of Automatic Program Synthesis Systems

Todd A Gross
Thomas A Nartker*
Department of Computer Science and Electrical Engineering
University of Nevada, Las Vegas

January 19, 1988

Report CSR-88-02



*Supported by the U. S. Army Research Office under Grant DAAL03-87-G-0004

A Classification of Automatic Program Synthesis Systems

Todd A. Gross
Thomas A. Nartker
Dept. of Electrical Engineering and Computer Science
University of Nevada, Las Vegas

January 19, 1988

1 Introduction

A great deal of attention has been paid lately to improving the process of developing software. Most of this attention has been directed to the development of *tools* to perform tasks for the user (e.g. editors, debuggers) and *environments* to integrate these tasks. This paper, however, is about another method of streamlining software development: creating programs to generate the software for us. That is, we tell the program what type of software we want to generate and it generates the software for us. This process has been given many names: automatic programming [2,7,25], program generation [14,29], synthetic programming [16], program construction [4,37], program writing [28], metaprogramming [27] and autoprogramming [9] among several. In this paper, we shall refer to this process as *automatic program synthesis*, or APS.

In Section 2 we define the term automatic program synthesis. Section 3 gives a set of classifications of various APS systems. Section 4 gives the authors' conclusions about existing APS systems and about the field of automatic program synthesis in general. Section 5 comments on the related yet distinct area of application generation.

2 What is Automatic Program Synthesis?

It is important that we have a clear definition of automatic program synthesis, because there are several types of systems that perform tasks similar to those outlined in this paper. For instance, compilers accept descriptions of what we want in the form of higher-level programs and produce software in the form of machine code. We do not consider compilers to be APS systems, because all necessary information about the problem, including a step-by-step solution,

is provided by the user. It is worth noting, however, that many authors see higher-level languages as in the realm of automatic program synthesis, see for instance [6,14,23].

It is easier to clarify what we mean by breaking the term "automatic program synthesis" into its three components. **Automatic** means with the aid of the computer. Clearly, computers must play a role in the creation of programs, but perhaps less clearly, they cannot perform the entire task for us. The part of the task they *do* perform is called *synthesis*, and will be defined later.

A **program**, for the purposes of this paper, is the object we generate using our APS system. It is a series of statements that, when executed, produce the desired output given all necessary inputs. Usually, the program is written in an executable higher-level language, although some papers [15,31] used a nonexecutable algorithmic language.

Synthesis is the process of creating code by computer. Many different types of programs synthesize code—including compilers, editors, optimizers, and decision table systems (discussed in [14]). In APS systems, code is generated from a *nondeterministic* description of the program. Which means that the description allows for *several* possible programs to be generated. Put another way, the description says *what* we want the generated program to do, not *how* we want the program to do it. The description, being nondeterministic, cannot be a program itself—it must be transformed into a program by the APS system.

We now define automatic program synthesis as the generation by the computer of a program (or programs) given a nondeterministic description of the program(s). All systems mentioned in the remainder of this paper (excepting those described in [14]) conform to this definition.

3 Classifying APS Systems

One can view the process of automatic program synthesis from two perspectives: that of the person who designed the APS system, and that of the person who uses it. From the user's perspective, the APS system is basically a black box: one gives it a description of the program one wants generated, and after some calculation it returns the desired program. Thus from the user's perspective, the most important aspect of the APS system is how one describes the program to the system. Formally, we call the program description a *specification*.

From the designer's perspective, the system is a good deal more complex. There are two major components of the APS system: the knowledge base and the code generator. The knowledge base contains information about how to operate on the specifications, and the code generator takes this knowledge and the specifications and generates the program. Note that this is by no means a clean division—in some cases the "knowledge base" is basically a set of predicates buried inside the code generator [10], in others the "code generator" is basically the actions taken by the knowledge base [30]. Still, it's an important distinction

to make, because the APS system designer can decide how to interconnect the two components to best suit the type of problem he or she wishes to solve.

3.1 Classification of Specifications

There are several basic means of specifying one's problem—logical predicates, input-output pairs, program traces, and natural language sentences are the most common.

Logical predicates are the oldest [39] and most common [13,15,16,20,21,22, 26,30,31,35,36,39] means of specifying inputs to APS systems. Typically, the specification would look something like:

$$(\forall i \in \mathrm{D})[f(i,o) \Leftrightarrow p(i,o) \wedge o \in \mathrm{R}]$$

where f is the program we want to create, i and o are the input and output to the program, D and R are the domain and range of the program, and p is a first-order logic sentence using bound variables i and o. Different systems, however, will use somewhat different logical forms. For instance, in [15], the universal quantifier is assumed and thus left out. In DEDALUS (in [30]), the specification structure is less formulaic and instead of $f(i,o)$ being a relation between the input and output we have $o = f(i)$ where f is a function on the input. These formulae are logically equivalent. As an example of a logical specification, we give one for finding the maximum element in a list:

$$(\forall \mathrm{L} \in \mathcal{L})[\mathbf{max}(\mathrm{L}, m) \Leftrightarrow m \in \mathrm{L} \wedge (\forall e \in \mathrm{L})[m \geq e] \wedge m \in \mathbf{Z}]$$

where $\mathcal{L}$ is the set of all lists of integers, $\mathbf{Z}$ is the set of all integers, and m is the maximal element of the list.

Input-output pairs have been used in fewer systems [10,11,37], despite their relative simplicity. One merely provides a set of inputs to your desired solution and the corresponding outputs. For instance, in the case of **max** we might give:

```
() -> nil          (1 2) -> 2
(1) -> 1           (2 1) -> 2
⋮                  ⋮
```

where the left side of the arrow has our input list and the right side our desired output.

Specification by program trace has been used in a few systems: autoprogrammer [8,9], I'C [33], and PSI [21,22]. autoprogrammer must be given a step-by-step execution of statements, but I'C and PSI need only be given snapshots of memory and conditional tests. As an example, we'll do a sample run of **max** on the list (2 1) as autoprogrammer would do it.

```
store   (2 1)      L          car    L        e
store   0          m          note   e   <    m
car     L          e          cdr    L        L
note    e     >    m          note   L   =    nil
store   e          m          stop
cdr     L          L
```

where **store** X Y assigns the value of X to Y, **car** X Y assigns the car of X to Y (and similarly for **cdr**). **note** doesn't perform any operation, but informs the program synthesizer that we tested a condition. Actually the example given wouldn't work in autoprogrammer—that system only performs numerical computations. Nonetheless, the formats are very similar. It should be noted that autoprogrammer can induct much of the program for us, saving the user the effort of specifying these parts. Memory trace systems can't do this at present, as it is far more difficult to induce a pattern from a set of memory snapshots. As one must do some programming oneself, program traces are an *imperative*, rather than a *declarative* specification.

Natural language specifications are straightforward enough: one uses English sentences to describe the problem. Naturally, one is restricted to a predetermined subset of English--one is also usually restricted to a specific problem domain. For more information, consult [25].

3.2 Classification of Program Synthesis Methods

As stated before, there are two aspects of synthesizing a program from a set of specifications: that of having a body of knowledge for operating on the specifications (which we labelled a "knowledge base"), and a system for using that information to generate code (the "code generator"). The knowledge base for an APS system is closely related to the method of specification, in the same way that parsers are closely related to lexical analyzers - just as parsers convert tokens to syntactic structures that are later converted to code, so our knowledge base converts specifications to procedural structures that are later converted to code. Therefore, we list each specification method and discuss the corresponding knowledge base:

Logic statements The knowledge base contains logical axioms like

$$f(x) \Leftrightarrow (f(x) \wedge g(x)) \vee (f(x) \wedge \neg g(x))$$

and axioms of nonlogical constructs like

$$(\exists x \in e.L)[f(x)] \Leftrightarrow f(e) \vee (\exists x \in L)[f(x)]$$

where $e.L$ and L are lists of elements.

Input-output pairs The knowledge base contains rules and information that enable us to induct on the examples. That is, it allows us to apply the rules uncovered in the examples to more general inputs.

Program traces The knowledge base contains rules on constructing generic tests and code constructs (loops, branches).

Natural language The knowledge base contains rules for converting human language into an internal unambiguous representation.

Classifying code generation techniques is more difficult. In some cases, the method of code generation is determined by the knowledge base but in others it isn't. For instance, if we use some sort of semantic network to represent the desired program (as in [4,25,28]), the code generator will clearly need to transform semantic structures into program statements. However, if we've represented our knowledge in the form of logical axioms, we can either store these in a data base and invoke them when they match our partially-derived specifications (as in [30]) or else we can embed these axioms in the code generator itself (along the lines of [10], but more like the decision table system as in [14]).

Basically, all methods of code generation involve *translation*, from an internal representation of the problem to a form a compiler can utilize. If either representation scheme subsumes the other, the translation can be purely syntactic. For instance, in most logic-oriented systems [13,15,16,26,30,35], the code generator sees both the final derivation of the specification and the code it is to construct as a string of tokens. Thus, one need only replace tokens with tokens. But if neither representation can subsume the other, there must be some semantic translation. For example, in most "knowledge-based" APS systems [1,4,25,28], the internal representation is a semantic network. Transformation of a semantic network to a program is nontrivial—indeed, most systems use heuristics to perform the translation rather than provably correct algorithms. Likewise, program trace systems internally represent the problem as a flow diagram, which also requires syntactic translation to generate linear code. Thus, it would appear that syntactic versus semantic translation is a fundamental classification. But there are other equally important but less universal means of classifying synthesis methods:

- Several systems use an internal knowledge base in the process of generating code. For instance, PSI has an internal model system that serves as intermediate code between the user's specification and the code generation. PROTOSYSTEM-I has several internal languages. But logic-oriented systems in general have no such interlingua.

- Some synthesis algorithms approach the problem from a nonstandard perspective. For instance, TRINS [38] uses an AI approach to choose between rules. programwriter incorporates an idea list in the course of generating problems. Novel ideas will require novel means of classification.

4 Conclusions

Having classified the existing methods of automatic program synthesis, we can now make some conclusions about APS systems in general. First of all, it is clear there is still much work to be done. In most cases, the programs generated were ones a competent programmer would have little difficulty constructing. As Summers says in [37]: "No paper on automatic LISP programming is complete without . . . the program **reverse**." Reversing a list of elements is typically one of the first recursive programs a student is taught. Systems with input-output pair specifications (like [37]) tend to restrict themselves to *transfer* problems—that is, problems where the input is copied to the output without any modification. Some systems, like programwriter [28] and PSI [4], have attempted realistic problems, unfortunately the authors don't know how successful they were or what range of problems they attempt to solve. There is relatively little available commercially, and even what is out there has been slow to be accepted [14]. Some companies, like AT&T [19] and Schlumberger [3], are developing their own APS systems.

It is also clear that different methods have their advantages and disadvantages. For instance, logic-oriented systems can be used for any computable problem and involve mostly syntactic transformations (which are relatively easy to generate). However, the specifications are difficult to produce correctly for nontrivial problems (although systems like DEDALUS are more approachable). Natural language systems are easier to give specifications in, but require vast amounts of information to decode the specifications, thus the problem domains are usually quite limited. A lot of work has been done in both these areas, and there appears to be no consensus on which specification method should be used.

Despite a lack of commercially sound results, there are many good reasons for developing APS systems:

- The cost of software is rising, especially in relation to the cost of hardware [12]. Therefore, it makes sense to use hardware to develop software.
- Modifying a program can be done by modifying the specification and rerunning the APS system, which is a good deal cleaner than splicing in code.
- The generated program is portable, and can be linked to already existing libraries. Tools and environments, in general, cannot.
- In some systems [16,28], one can develop several interrelated programs at one time.
- In many cases ([9,24] and any input-output system), one can provide a partial specification and the system will either make assumptions about the rest or prompt for more information (or both).

- Especially with logic-oriented systems, the software is provably correct.

For these reasons and others, it is important that we continue to research the development of provably correct and commercially sound APS systems. The reader is referred to [5,6,7] for more information on specific APS systems and methods.

5 Final Remarks

In writing this paper, the authors focussed their attention on a relatively restricted set of systems, namely those that generated complete programs with no foreknowledge of the program desired. That is to say, systems that generated programs knowing only the inputs to the program, the desired outputs to the program, and one or more techniques of creating code that will generate the desired outputs from the inputs. Several of these systems had a restricted domain (NLPQ, for instance, restricts itself to programs that simulate servicing a queue of customers [25]), but the knowledge of the domain does not include predefined methods of generating programs that solve problems under that domain.

In choosing to restrict our attention this way, the authors neglected the highly commercial and productive area of *application generation*. Although many systems mentioned by Cardenas [14] would fall under this category, of more interest are *screen* and *report generators*, particularly as elements of so-called fourth generation language (4GL) systems. These generators use standard text formatting routines (much like **curses** in the UNIX operating system), but are incorporated with a database management system. This saves a lot of effort in generating customized interfaces between the user and the application, and is therefore highly valuable.

Generally speaking, an *application generator* creates programs that perform specific tasks using well known, optimized (i.e. precoded) techniques. The systems in Cardenas' paper produce source code, most 4GL systems produce object code. Application generators share many of the same valuable characteristics of program synthesizers—including portability, ease of modification, and greater use of hardware to generate software. And they are available for use the general public. But on the downside, specifications are either programs (in the case of 4GL systems) or answers to prerestricted questions (in the case of decision table/questionnaire/customizing systems). Further, they can only be of limited use in generating programs that we have no apparently optimal solution for. One can see why research has concentrated on synthesis of programs, rather than generation.

Of course, there has been a good deal of research in program generation, particularly generation of parsers and compilers [32,34]. Interesting research in program generation has been done even in recent years, in areas as diverse as Gaussian elimination [3] and generic user interfaces [17,18]. While many

of these generators work in less understood problem domains using less understood algorithms than application generators—and indeed less understood domains than many APS systems work under—we always know in advance what programs will be generated, barring errors in the design and/or implementation of the generator. And this makes the generator a tool rather than an object of research.

References

[1] Automatic Programming Group. *Project MAC Progress Report X*. Technical Report, Massachusetts Institute of Technology, Cambridge, MA, Jul 1973. Page 172-191.

[2] R Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257-1268, 1985.

[3] D Barstow, R Duffey, S Smoliar, and S Vestal. An automatic programming system to support an experimental science. In *Proceedings of the Sixth International Conference on Software Engineering*, pages 360-366, Sep 1982.

[4] D R Barstow. *Knowledge-Based Program Construction*. North-Holland, New York, 1979.

[5] A Biermann, G Guiho, and Y Kodratoff. *Automatic Program Construction Techniques*, chapter 1. Macmillan, New York, 1984. titled An Overview of Automatic Program Construction Techniques.

[6] A W Biermann. *Advances in Computers*, pages 1-63. Volume 15, Academic Press, New York, 1976.

[7] A W Biermann. Automatic programming: a tutorial on formal methodologies. *Journal of Symbolic Computation*, 1:119-142, 1986.

[8] A W Biermann, R I Baum, and F Petry. Speeding up the synthesis of programs from traces. *IEEE Transactions on Computers*, C-24(2):122-136, 1975.

[9] A W Biermann and R Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, SE-2(3):141-153, 1976.

[10] A W Biermann and D R Smith. A production rule mechanism for generating LISP code. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-9(5):260-276, 1979.

[11] T J Biggerstaff. *C2: A 'Super Compiler' Approach to Automatic Programming*. PhD thesis, University of Washington, Seattle, WA, Jan 1976.

[12] B W Boehm. Software and its impact: a quantative assessment. *DATAMATION*, 48–59, May 1973.

[13] R M Burstall and J Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[14] A F Cardenas. Technology for automatic generation of application programs—a pragmatic view. *MIS Quarterly*, Sep 1977.

[15] K L Clark and S Sickel. Predicate logic: a calculus for deriving programs. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 419–420, Aug 1977.

[16] N Dershowitz. Synthetic programming. *Artificial Intelligence*, 25(3):323–373, 1985.

[17] P Dewan. *Automatic Generation of User Interfaces*. PhD thesis, University of Wisconsin, Madison, WI, Aug 1986.

[18] P Dewan and M Solomon. Dost: an environment to support automatic generation of user interfaces. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Symposium on Practical Software Development Environments*, pages 150–159, Dec 1986.

[19] S L Ehrenreich and W A Harris. JMOS: stepping outside with new cost control. *Bell Laboratories Record*, Jul 1985.

[20] R Follett. Describing the complete effects of programs. In *Proceedings of the Symposium on Language Design and Programming Methodology*, pages 95–104, Sep 1979.

[21] C C Green. The design of the PSI program synthesis system. In *Proceedings of the Second International Conference on Software Engineering*, pages 4–18, Oct 1976.

[22] C C Green. A summary of the PSI program synthesis system. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 380–381, Aug 1977.

[23] M Hammer and G Ruth. *Research Directions in Software Technology*, chapter 20. MIT Press, Cambridge, MA, 1979. titled Automating the Software Development Process.

[24] G Heidorn. *Natural Language Inputs to a Simulation Programming System*. Technical Report, Naval Postgraduate School, Monterey, CA, 1972.

[25] G E Heidorn. Automatic programming through natural language dialogue: a survey. *IBM Journal of Research and Development*, 20(4):302–313, 1976.

[26] R C Lee, C L Chang, and R J Waldinger. An improved program-synthesizing algorithm and its correctness. *Communications of the ACM*, 17(4):211–217, 1974.

[27] L S Levy. A metaprogramming method and its economic justification. *IEEE Transactions on Software Engineering*, SE-12(2):272–277, 1986.

[28] W J Long. *A Program Writer*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1977. Report LCS/TR-187.

[29] P A Luker and A Burns. Program generators and generation software. *Computer Journal*, 29(4):315–321, 1986.

[30] Z Manna and R Waldinger. Synthesis: dreams ⇒ programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, 1979.

[31] Z Manna and R Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[32] L Paulson. *A Compiler Generator for Semantic Grammars*. PhD thesis, Stanford University, Stanford, CA, Dec 1981.

[33] F E Petry. *Program Inference from Example Computations Represented by Memory Snapshot Traces*. PhD thesis, Ohio State University, Columbus, OH, 1974.

[34] S P Reiss. Automatic compiler production: the front end. *IEEE Transactions on Software Engineering*, SE-13(6):609 627, 1987.

[35] D R Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[36] P A Subrahmanyam. An automatic/interactive software development system: formal basis and design. In A Wasserman, editor, *Proceedings of the IFIP WG8.1 Working Conference on Automated Tools for Information Systems Design and Development*, pages 125–146, North-Holland, 1982.

[37] P D Summers. *Program Construction from Examples*. PhD thesis, Yale University, New Haven, CT, Dec 1975.

[38] V Vojtek, L Molnar, and P Navrat. Automatic program synthesis using heuristics and interaction. *Computers and Artificial Intelligence*, 5(5):429–442, 1986.

[39] R Waldinger and R Lee. PROW: a step toward automatic programming. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1969.

Appendix A List of APS Systems

This is a list of some of the major automatic program synthesis systems, all developed in the past 20 years. The systems are given in approximately chronological order, and include the name and author of the system, a brief description of the method of input to the system, the knowledge base (KB) representing all information used to synthesize programs, and the method of code generation (CG). In those cases where the method of code generation couldn't be determined from the papers read, a question mark is given.

PROW
R J Waldinger

Input: Conditional relation

KB: Waldinger-Lee algorithm

CG: Logical manipulation via the Waldinger-Lee Algorithm

NLP(Q)
G Heidorn

Input: A subset of natural language, on a specific domain (serving objects that must wait in line).

KB: Rules for converting natural language to a semantic network, knowledge of the problem domain, logical/mathematical knowledge

CG: ?

PROTOSYSTEM-I
Project MAC

Input: English statements converted to MAPL

KB: Translation rules, rules for handling conventional program features

CG: Simulation of MAPL statements, with suggestions for improvement in case of failure

I'C
F E Petry

Input: Memory trace from a sample computation, list of functions, variables and conditions used in the sample computation

KB: Rules for finding potential statements, for ordering candidates by likelihood of working, for finding the minimum program size

CG: Take the set of candidate statements, attempt to create a minimal flow diagram. Backtrack to previous statements if we can't get a proper program. If no program is possible, increase the number of statements allowed

autoprogrammer
A W Biermann, R Krishnaswamy, R I Baum, F E Petry

Input: Partial *in situ* traces of the program

KB: Generation of optimal flow diagrams, condition testing, induction of loops

CG: Transformation to flow diagrams

THESYS
P D Summers

Input: A set of example input-output pairs

KB: Rules for induction on list elements

CG: Convert outputs to elements of input list, look for a pattern. If one can't be found, try introducing a variable

C2
T J Biggerstaff

Input: Sample input-possible output pairs

KB: Algorithm for creating a function strategy tree, generalizing to a control graph, and reformatting to a LISP program. Logical manipulation of function requirements

CG: Generation of the function strategy tree, control graph, and mapping the graph to a LISP program

PSI
C Green et al

Input: Natural language sentences or program traces

KB: A set of interacting modules, converters/translators from internal form to internal form; property, query and refinement rules (in PECOS)

CG: Two interacting modules—a code generator and an efficiency expert. The code generator applies rules of refinement to code, establishes properties, and tests for patterns under a prespecified heuristic agenda. The efficiency expert examines potential algorithms for optimality in space-time efficiency

programwriter
W J Long

Input: OWL-I specifications of input and output

KB: METHODs, SCHEMAs, INTENTs, IDEAs, DEFINITIONs; various information about how to achieve subgoals, and world knowledge divided into 5 predefined interacting models—domain, argument passing and control, data, input/output, and target language

CG: Two phases: an analyze/plan loop phase followed by a coding phase. The analyze/plan loop orchestrates the used of METHODs, etc., to modify the semantic network that represents the program under development. Goals are invoked from a GOAL list, METHODs that fit are used to solve for the goal. SCHEMAs are used to corroborate disjoint goals. IDEAs are culled from an idea list when current METHODs fail. When the analyze/plan loop generates the desired program set (can be ¿1 program), the coding phase constructs a LISP program

MODEL II
N S Prywes, A Pneuli, S Shastry

Input: Data structure and assertion-of-values specifications

KB: Rules on manipulation of inputs, generation of correct nested loops

CG: ?

Production Rule Mechanism
A W Biermann and D R Smith

Input: A single input-output specification in list form

KB: Abstract pattern matching rules, rules for conditional construction

CG: Generation of lambdas via application of the pattern matchers

DEDALUS
Z Manna and R Waldinger

Input: First-order logic sentences with syntactic sugar

KB: Rules of first-order logic, conditionality, recursion, etc.

CG: Logic-oriented syntactic transformation

PROSYN
R Follett

Input: First-order logic input-output specifications

KB: Rules for deriving passback pairs, for protecting already achieved goals

CG: Generate the most specific pregoal for each goal, backtrack if we reach an unsolvable goal

CYPRESS
D R Smith

Input: Logical input-output assertion of the form $(\forall x \in D)(\exists z \in R)[I : x \Rightarrow O : (x, z)]$

KB: RAINBOW, which generates the best antecedent for a specification, algorithm for construction of divide-and-conquer algorithms, knowledge of specific domain and range types

CG: Application of the most specific antecedent to the divide-and-conquer algorithm, backtrack if it fails

Synthetic Programming
N Dershowitz

Input: An output goal and a set of input and output assertions in a template form

KB: Translational rules for creating desired output constructs

CG: Rule application with backtrack

TRINS
V Vojtek, L Molnar, and P Navrat

Input: Input/output specifications in the form compute:f(I) where:R(I)

KB: A set of potential transformation rules, heuristic weighing algorithms for rules

CG: For a (sub)computation, find all applicable transformations, weigh each rule, and pick the one with the lowest weight. If there's a condition, partition the input space

END

DATE

FILMED

DTIC

JULY 88